J. Symbolic Computation (1991) **12**, 275–298

A Methodology for Detecting Shared Variable Dependencies in Logic Programs

B. SCHEND

BASF AG, ZXT-Informatik-Technologie, 6700 Ludwigshafen, West Germany

(*Received* 4 *December* 1989)

AND-parallel execution of logic programs turns out to be an intricate matter whenever clause literals are linked by shared variables. Shared variable dependencies call for special precautions to prevent processes from computing inconsistent bindings while working on clause literals simultaneously. Therefore, most systems restrict AND-parallelism to its producer/consumer style by allowing only independent literals to run in parallel. These literals then act as producers of variable bindings which will be consumed by dependent literals in subsequent steps. However, to implement producer/consumer parallelism efficiently, appropriate methods must be available for detecting dependencies caused by shared variables. Concerning such dependencies this paper presents a methodology which may serve as a basis for efficient dependency checking by performing compile and run-time analysis of program clauses.

1. Introduction

The development of parallel execution strategies is turning out to be one of the most promising ways of not only improving run-time behaviour of logic programs, but also allowing a greater degree of declarative programming. Obviously, freeing a programmer from the von Neumann style of programming requires a system which is capable of executing programs without relying on any external support. A parallel system, therefore, must be able to detect and execute parallelism automatically including, for instance, determination of shared variable dependencies and selection of producer literals (Conery, 1983). This of course should not prevent a programmer from using a limited amount of syntactical tools—like annotating variables—to tune his program for reasons of efficiency.

The attractiveness of parallelism has manifested itself in a great number of models and systems which have been worked out and published. This in return has been going hand in hand with an investigation of various execution strategies ranging over co-routining, pipelining, dataflow and process based evaluation methods (Clark & Gregory, 1984; Tick & Warren, 1984; Halim, 1986; Conery, 1987). Especially using processes for realizing parallelism is characteristic of many models proposed so far. Within these models program execution is typically settled on a set of processes communicating via messages while running on a multiprocessor architecture.

Altogether, process based as well as other parallel systems have to deal with two basic kinds of parallelism. *AND-parallelism* on one side results from solving clause literals simultaneously, while *OR-parallelism* is achieved by executing all the matching clauses of a literal in parallel. In a process system the first type of parallelism is controlled by *AND-processes* whereas the second one is managed by *OR-processes*. According to the problem reduction principle (Kowalski, 1979) AND-processes solve their task by creating descendant OR-processes for every literal to be executed in parallel. OR-processes in

0747-7171/91/090275+24 $03.00/0

return start AND-processes for all clauses matching the literal at hand. Whereas the task of an OR-process is known to be relatively easy, AND-processes have to deal with considerably more problems, which mainly stem from shared variable dependencies among clause body literals. If descendant OR-processes work on dependent literals simultaneously, they are likely to compute inconsistent variable bindings. To get a better idea of inconsistencies let us observe some fictitious system when executing the first clause of the example program shown in Figure 1.

At the very beginning of clause execution our system initiates an AND-process $\mathcal{P}$ to compute solutions for $p(x, y) \leftarrow q(x), r(x, y)$. For the sake of simplicity we assume head variables x and y to have no values at this time. The AND-process subsequently creates descendant OR-processes $\mathcal{P}_{q(x)}$ and $\mathcal{P}_{r(x,y)}$ to work on $q(x)$ and $r(x, y)$ in parallel. On the basis of our example program process $\mathcal{P}_{q(x)}$ computes the answer $\sigma = \{x/f(u)\}$, i.e. "x bound to term $f(u)$", while $\mathcal{P}_{r(x,y)}$ finds $\gamma_1 = \{x/b, y/c\}$ and $\gamma_2 = \{x/f(a), y/c\}$. All answers are sent to the AND-process via messages. Now, having received these partial solutions from its OR-sons the AND-process must join them together to produce a complete solution for both, $q(x)$ *and* $r(x, y)$. By doing so it finds σ consistent with γ_2 but inconsistent with γ_1; inconsistency of σ and γ_1 is due to non-unifiability of bindings for variable x. After all, γ_1 does not contribute to a complete solution and thus was computed without yielding any profit. Only joining σ and γ_2 gives a useful solution which is $\{x/f(a), y/c\}$.

In general, joining partial solutions is a time-consuming task whenever answer sets of OR-processes are very large. The effort required could be justified on principle if most of the answers were consistent with each other. Yet, in practice many answers show no profit in the sense mentioned above. Computation of such answers, however, can go along with numerous redundant resolution steps and, moreover, creation of many superfluous processes. Especially spawning processes promiscuously may affect program execution seriously on account of limited processor resources. Based on this observation most parallel systems realize AND-parallelism in a producer/consumer style by restricting it to literals which have no variables in common and thus are independent from each other. No inconsistencies will arise among solutions computed in parallel since a shared variable is allowed for only one producing OR-process at a time. Of course, in order to realize this type of parallelism AND-processes have to check dependencies before creating OR-processes to solve body literals in parallel. Although dependency checking has to be performed very often during clause execution—especially when literal ordering is done dynamically (Conery, 1987)—most systems tackle the checking problem in a superficial manner only. In this regard a general framework for effective dependency checking will be presented here. Based on a compile and run-time analysis of programs it may be used to derive efficient methods and algorithms for determining dependencies in a parallel execution environment.

The next section introduces some notational conventions used throughout this paper. Upon that literal dependencies will be studied from a run-time as well as compile-time point of view. In particular, ways and means are shown for extracting information about

(1) $p(x, y) \leftarrow q(x), r(x, y)$
(2) $q(f(u)) \leftarrow$
(3) $r(b, c) \leftarrow$
(4) $r(f(a), c) \leftarrow$

Figure 1. Example program.

dependencies on a pure syntactical level, which will serve as a guide for efficient dependency checking at run time. In section 4 dependency graphs are introduced allowing not only an adequate representation of syntactical dependencies, but also an effective computation of independent literals at execution time. In this connection dependency checking will be shown in the form of graph reduction. The succeeding sections then present the concepts of binding patterns and test collections, which are used to characterize the minimal amount of binding tests required for reducing a graph completely. Dependency graphs, binding patterns and test collections will be essential parts of our dependency checking methodology. Finally, an example algorithm for reduction of dependency graphs will be developed.

2. Notation

For the rest of the paper the reader is assumed to be familiar with the basic terminology of logic programming as introduced by Lloyd (1984). Some supplementary notation used subsequently is given below.

The set of variables is denoted by **V**, the elements of which will typically be named by letters x, y, z, u and v. Letters a, b, c and d denote constants, and f, g, h function symbols. An *expression* can be a term, literal, or list of both. If e represents an expression, $\mathrm{Var}(e)$ is the set of all variables occurring in e. Substitutions are named by Greek letters like θ, σ and γ; a substitution $\theta = \{x_1/t_1, \ldots, x_n/t_n\}$, $n \geq 0$, binds variable x_i to the term t_i, $1 \leq i \leq n$. Two expressions e_1 and e_2 are *dependent* if $\mathrm{Var}(e_1) \cap \mathrm{Var}(e_2) \neq \emptyset$; e_1 and e_2 are called *connected* by a substitution θ if $\theta(e_1)$ and $\theta(e_2)$ are dependent. *Composition* $\theta_1 \circ \theta_2$ of two substitutions is defined as $\theta_1 \circ \theta_2(e) = \theta_1(\theta_2(e))$ for every expression e. Let $M = M_1 \times M_2$ be the Cartesian product of two non-empty sets M_1 and M_2. Then $[M]_i = M_i$ and $[m]_i = m_i$ for $m = (m_1, m_2) \in M$ and $i \in \{1, 2\}$.

3. Shared Variable Dependencies

One way to support detection of shared variable dependencies is to analyse literals already syntactically for extracting some kind of information, which finally contributes to a fast determination of dependencies in an actual binding environment. Analysing literals on a pure syntactical level suggests a distinction of dependencies, namely syntactical dependency on one side and run-time dependency on the other.

DEFINITION 3.1. Two literals L_1 and L_2 are called *run-time dependent* wrt a substitution θ if L_1 and L_2 are connected by θ. We write $L_1 \underline{\mathrm{rd}}_\theta L_2$ for short.

Whereas the run-time type of dependency is closely associated with an actual environment, its syntactical counterpart will solely be founded on properties which are *not* related to any special bindings. As will become clear, these properties are based on a classification of clause variables giving them either a global or local character. Classification itself always happens in connection with literal configurations.

DEFINITION 3.2. Let Γ and Π be sets of clause literals. Then $\mathscr{C} = [\Gamma | \Pi]$ is called a (*literal*) *configuration* with *head* Γ and *body* Π.

To point out the role of literal configurations let us execute clause $p(x, y) \leftarrow q(x, y)$, $r(x)$, $s(x)$ within environment $\theta_0 = \{x/f(u)\}$. As usual, clause execution is settled on an AND-process $\mathcal{P}$, which initially finds itself in *start configuration* $\mathcal{C}_0 = [\{p(x, y)\} | \{q(x, y), r(x), s(x)\}]$. First of all, process $\mathcal{P}$ determines dependencies among body literals, i.e. it checks whether L and L' are run-time dependent wrt θ_0 for each pair (L, L') of distinct body literals. Since all literals turn out dependent, only one of them, say $q(x, y)$, can be selected as a producer literal. Now, making $q(x, y)$ a producer results in a new configuration $\mathcal{C}_1 = [\{q(x, y)\} | \{r(x), s(x)\}]$, which we call the *successor* of $\mathcal{C}_0$. Upon entering the new configuration our AND-process $\mathcal{P}$ creates a descendant OR-process $\mathcal{P}_{q(x,y)}$ to solve literal $q(x, y)$ in the environment θ_0, i.e. to compute answer substitutions for goal $\theta_0(q(x, y)) = q(f(u), y)$. Let $\mathcal{P}_{q(x,y)}$ find the answer $\gamma = \{u/a, y/g(v)\}$. Having received that substitution our AND-process produces a new environment $\theta_1 = \gamma \circ \theta_0 = \{x/f(a), y/g(v)\}$. As θ_1 binds variable x to a ground term, all body literals of $\mathcal{C}_1$ are *not* run-time dependent wrt θ_1 and thus can be solved in parallel. Our next configuration therefore is $\mathcal{C}_2 = [\{r(x), s(x)\} | \varnothing]$ having θ_1 as its actual environment; configuration $\mathcal{C}_2$, which from the point of $\mathcal{C}_1$ plays the role of a consumer, is also called a *final configuration* since there are no body literals left for execution. Like before, OR-processes $\mathcal{P}_{r(x)}$ and $\mathcal{P}_{s(x)}$ start to compute answers for $\theta_1(r(x)) = r(f(a))$ and $\theta_1(s(x)) = s(f(a))$, respectively. Produced substitutions of $\mathcal{C}_2$ finally result in solutions for our initial program clause.

The example above has shown producer/consumer parallelism to be a matter of creating and executing literal configurations. Thereby, OR-processes compute answers for every head literal of the configuration at hand. Then these answers are used to produce new environments, within which body literals will be checked for dependency. After dependency checking independent literals are selected to obtain the head of the succeeding configuration. Altogether, applying this procedure repeatedly results in a configuration sequence $\mathcal{C}_0, \ldots, \mathcal{C}_n$ where $\mathcal{C}_0$ is the start and $\mathcal{C}_n$ is the final configuration. Each configuration $\mathcal{C}_i$ *produces* substitutions being *consumed* by its successor $\mathcal{C}_{i+1}$, $1 \leq i < n$. Produced substitutions of $\mathcal{C}_n$ finally represent complete solutions for the clause at hand.

DEFINITION 3.3. Let $\mathcal{C} = [\Gamma | \Pi]$ be a literal configuration consuming substitution θ. Further, let $\Gamma = \{L_1, \ldots, L_n\}$, $n \geq 1$, and $\mathcal{S}_1, \ldots, \mathcal{S}_n$ be the sets of answer substitutions computed for literals in Γ, i.e. $\mathcal{S}_j$ contains all answers found for θL_j, $1 \leq j \leq n$. (Note, all literals in Γ are run-time independent wrt θ!)

Then $\mathcal{PS}(\mathcal{C}, \theta) = \{\text{Join}(\gamma_1, \ldots, \gamma_n) \circ \theta \mid \gamma_j \in \mathcal{S}_j, 1 \leq j \leq n\}$ is the set of *produced substitutions* of $\mathcal{C}$, where $\text{Join}(\gamma_1, \ldots, \gamma_n) = \gamma_1 \cup \cdots \cup \gamma_n$. For a start configuration $\mathcal{C}_s$ we simply define $\mathcal{PS}(\mathcal{C}_s, \theta) = \{\theta\}$.

A produced substitution of some configuration $\mathcal{C}$ always is a consumed one from the point of its successor, which itself views it as a *calling* environment. Given such an environment, a consumer immediately tries to compute produced substitutions of its own. In this regard, should a configuration fail to produce any substitution, backtracking is initiated to its predecessor for getting a new calling environment (Schend, 1989). Each time backtracking occurs the producing configuration takes a new element from its set of produced substitutions, and upon that, checks body literals again for dependency. Checking eventually results in different dependencies and thus in creating a new successor configuration. Besides, backtracking is also initiated after all possible solutions have been produced within the actual environment. With backtracking mechanism in mind we now

turn back to the syntactical type of dependency by defining global and local variables of configuration bodies.

DEFINITION 3.4. Let $\mathscr{C} = [\Gamma | \Pi]$ be a literal configuration with consumed substitution θ. The set of *global variables* of body Π of $\mathscr{C}$ is inductively defined as follows:

(i) Every variable $x \in \mathrm{Var}(\Pi) \cap \mathrm{Var}(\Gamma)$ with $\mathrm{Var}(\theta(x)) \neq \varnothing$ is a global variable.

(ii) Every variable $y \in \mathrm{Var}(\Pi) - \mathrm{Var}(\Gamma)$ connected to a global variable by θ is also a global variable.

Variables of Π being not global are called *local variables.*

The intention behind defining a variable global is to point out that its present value may be changed by a produced substitution. In this connection part (ii) of Definition 3.4 takes into account that body variables may have been *aliased* with head variables in a previous execution step (DeGroot, 1984). By way of example, looking at configuration $[\{p(x, y)\} | \{q(x, z), r(x), s(y)\}]$ with consumed substitution $\theta = \{x/g(u)\}$ we can classify x and y as global variables, whereas z represents a local one. In case of $\theta = \{x/g(v), y/a, z/f(v)\}$, however, z turns out to be a global variable since connected to x. Observe, if $\mathscr{C} = [\Gamma | \Pi]$ is a start configuration, i.e. Γ corresponds to the head and Π to the body of some program clause, all variables of Π not occurring in Γ are local to Π. Produced substitutions of start configurations—they are the substitutions a clause is called with—never bind such variables (see Lloyd, 1984).

Looking back to Definition 3.1 (run-time dependency) we find that checking the dependency of two literals L_1 and L_2 within a produced environment θ practically means searching the instances $\theta(L_1)$ and $\theta(L_2)$ for common variables. As these instances may show very complex term structures (DeGroot, 1984), the expense of time required for term traversing cannot be ignored, all the more since checking has to be done repeatedly during execution of literal configurations. Performing dependency tests repeatedly suggests a kind of syntactical analysis in order to gain some dependency information not related to any special substitution.

Intuitively, body literals should be viewed syntactically dependent if they can become run-time dependent by some produced substitution. Analysing clause literals on a syntactical level, of course, does not enable us to foresee run-time dependency in general. However, what we will obtain thereby is some knowledge about dependencies, which may serve as a guide for efficient dependency checking each time a new substitution has been produced. The kind of information available on a syntactical level will now be characterized by inspecting some sample literal configurations. They all are assumed to consume $\theta = \varnothing$ as their actual calling environment.

In configuration $[\{p(x)\} | \{q(x), r(x)\}]$ both body literals $q(x)$ and $r(x)$ show syntactical dependency since linked via their global variable x. Yet this type of dependency does not imply run-time dependency of $q(x)$ and $r(x)$. Just imagine a produced substitution $\theta' = \{x/a\}$ which turns them independent by binding variable x to a ground term. We can only conclude in the above case of a configuration, that literals linked by some global variable might, but need not become run-time dependent. A similar situation can be met even if there are no variables in common. For instance, look at body literals of $[\{p(x, y)\} | \{q(x), r(y)\}]$, which share no variables but can become dependent at run time if values of x and y are connected by a produced binding environment, say $\sigma = \{x/u, y/f(u)\}$. So like before, run-time dependency solely rests with the actual bindings. Despite this, existence of a common local variable, like z in $[\{p(x)\} | \{q(x, z), r(z)\}]$

definitely leads to run-time dependency just because produced substitutions do not assign values to local variables.

It should be understood that the information available about dependencies on a syntactical level comes from global and local variables. So it is these variables which constitute our dependency information as defined below. In the following all literals, unless stated otherwise, are assumed to be body literals of some literal configuration. Consequently, their global and local variables are implicitly defined.

DEFINITION 3.5. Let L_1 and L_2 be literals with global variable set V.

(a) $\Delta_V(L_1, L_2) = (D, I)$ is called the *dependency information* of L_1 and L_2 with
$D = \mathrm{Var}(L_1) \cap \mathrm{Var}(L_2)$,
$I = \{(x, y) \mid x \in (\mathrm{Var}(L_1) \cap V) - D \text{ and } y \in (\mathrm{Var}(L_2) \cap V) - D\}$.
An *empty dependency information*, i.e. $D = I = \emptyset$, is denoted by $\Delta_\emptyset$.

(b) L_1 and L_2 are *syntactically dependent* if $\Delta_V(L_1, L_2) \neq \Delta_\emptyset$.

(c) Syntactical dependency provided, literals are called *strongly dependent* if the D-component of their dependency information contains a local variable, else they are called *weakly dependent.*

The D-component of a dependency information $\Delta_V(L_1, L_2)$ comprises all variables which are shared by both literals and therefore signal dependency *directly.* The I-component on the other hand consists of variables which, compared to variables in D, point to a dependency only *indirectly.* Strong dependency implies run-time dependency whereas the weak type does not. On the other hand, literals with an empty dependency information definitely stay independent at run time, i.e. syntactical implies run-time independency. Some examples of dependency information can be found in Table 1.

Dependency information of clause literals does not only supply knowledge about syntactical dependencies, but can also be interpreted as an algorithm for performing dependency checking at run time. By way of example, information $\Delta_V(L_1, L_2) = (\{v\}, \{(x, y)\})$ with $V = \{v, x, y\}$ can be transformed into the following Prolog-like program (θ represents a produced substitution).

$$\mathrm{dependent}(L_1, L_2, \theta) \leftarrow \mathrm{Var}(\theta(v)) \neq \emptyset,$$
$$\mathrm{dependent}(L_1, L_2, \theta) \leftarrow \mathrm{Var}(\theta(x)) \cap \mathrm{Var}(\theta(y)) \neq \emptyset.$$

Algorithms extracted from dependency information obviously are non-deterministic. The information above, for instance, does not tell us what rule to apply first, i.e. should we first check whether variable v is non-ground or values of x and y are connected? As will become clear, choosing the appropriate order is of great significance for achieving efficient dependency checking. But first of all it has to be shown how dependency information can be used to determine dependencies among body literals.

Table 1. Literals and their corresponding dependency information

L_1	L_2	V	$\Delta_V(L_1, L_2)$
$q(x)$	$r(x)$	$\{x\}$	$(\{x\}, \emptyset)$
$q(x)$	$r(y)$	$\{x, y\}$	$(\emptyset, \{(x, y)\})$
$q(x, z)$	$r(z, y)$	$\{x, y\}$	$(\{z\}, \{(x, y)\})$
$q(x, y)$	$r(x, y)$	$\{x, y\}$	$(\{x, y\}, \emptyset)$
$q(x)$	$r(y)$	$\emptyset$	$\Delta_\emptyset$

Table 2. Literals and their reduced dependency information

L_1	L_2	V	$\Delta_V(L_1, L_2)$	$\Delta_{V,\theta}(L_1, L_2)$
$q(x)$	$r(x)$	$\{x\}$	$(\{x\}, \varnothing)$	$\Delta_\varnothing$
$q(x)$	$r(y)$	$\{x, y\}$	$(\varnothing, \{(x, y)\})$	$\Delta_\varnothing$
$q(x, z)$	$r(z, y)$	$\{x, y\}$	$(\{z\}, \{(x, y)\})$	$(\{z\}, \varnothing)$
$q(x, y)$	$r(x, y)$	$\{x, y\}$	$(\{x, y\}, \varnothing)$	$(\{y\}, \varnothing)$
$q(x)$	$r(y)$	$\varnothing$	$\Delta_\varnothing$	$\Delta_\varnothing$

DEFINITION 3.6. Let $\Delta_V(L_1, L_2) = (D, I)$ be a dependency information and θ a produced substitution. The *θ-reduced dependency information* of L_1 and L_2 is denoted by $\Delta_{V,\theta}(L_1, L_2) = (D_r, I_r)$ with

$$D_r = \{x \in D \mid \mathrm{Var}(\theta(x)) \neq \varnothing\},$$
$$I_r = \{(x, y) \in I \mid \mathrm{Var}(\theta(x)) \cap \mathrm{Var}(\theta(y)) \neq \varnothing\}.$$

Some examples of reduced dependency information can be found in Table 2; the substitution used for reduction is $\theta = \{x/a, y/g(u)\}$.

There is a strong relationship between run-time dependency of literals and their reduced dependency information. Clearly two literals are run-time dependent if and only if their reduced dependency information is not empty, i.e. $L_1 \underline{\mathrm{rd}}_\theta L_2$ iff $\Delta_{V,\theta}(L_1, L_2) \neq \Delta_\varnothing$. As a consequence dependency checking may be turned into a matter of testing dependency information for being empty or non-empty through reduction.

Based on this observation detecting run-time dependency of two literals having (D, I) as their dependency information can be done by answering the following questions: Is there any non-ground variable in the D-component, or alternatively, can we find a pair of connected variables in the I-component? These questions indicates that we basically need two types of tests for checking dependency via reduction. A *ground test* takes a single variable and determines if its value is a ground term. A *connection test*, on the other hand, checks if two variables are connected. By the way, ground and connection tests together constitute what was called a *binding test* in a previous section. Note that ground tests are less expensive than their connection counterparts. Testing a variable for groundness means searching through a single term structure only. Performing connection tests, however, possibly requires traversing two terms and eventually computing intersection of variable sets. Fortunately, running connection tests explicitly often turns out to be unnecessary in practice. If, for instance, a variable x is already known to be ground by some previous test, it cannot be connected to another variable. Any connection test involving x would be redundant therefore. Due to this observation every algorithm for dependency checking should try to solve its task by performing ground tests only and avoiding expensive connection tests whenever possible.

Before developing such an algorithm, we need some mechanism to represent dependency information about an arbitrary number of clause literals. Remember that the concept of dependency information restricts our attention to pairs of literals. Measures to remedy this situation will be introduced in the next section.

4. Dependency Graphs

Information about syntactical dependencies can naturally be reflected by a special kind of graph, the nodes of which correspond to body literals of the configuration at hand,

with edges being labelled by their corresponding dependency information. Since these graphs represent dependencies of clause literals, they are called dependency graphs from now on. With respect to a formal definition, special terminology has to be introduced first.

DEFINITION 4.1.

(a) An (*undirected*) *graph* $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ consists of a finite set of vertices $\mathcal{V} = \mathcal{V}(\mathcal{G})$ and a set of edges $\mathcal{E} = \mathcal{E}(\mathcal{G})$. An edge is an unordered pair (set) of distinct vertices v_1 and v_2, denoted by $[v_1, v_2]$. We write $\mathcal{G} = \mathcal{G}_\varnothing$ if $\mathcal{V} = \mathcal{E} = \varnothing$.

(b) $\mathcal{G}$ is a *subgraph* of a graph $\mathcal{G}'$ if $\mathcal{V}(\mathcal{G}) \subseteq \mathcal{V}(\mathcal{G}')$ and $\mathcal{E}(\mathcal{G}) \subseteq \mathcal{E}(\mathcal{G}')$.

All prerequisites for representing syntactical dependencies have now been introduced. Again, for all literal sets below we assume local and global variables to be implicitly defined.

DEFINITION 4.2. Let Π be the body of some configuration with global variable set V. The *dependency graph* $\mathcal{G}$ of Π is a graph named $DG_V(\Pi) = (\mathcal{V}, \mathcal{E})$ such that

$$\mathcal{V} = \Pi,$$

$$\mathcal{E} = \{[L, L'] \mid L, L' \in \Pi,\ L \neq L' \text{ and } \Delta_V(L, L') \neq \Delta_\varnothing\}.$$

Each edge $[L, L'] \in \mathcal{E}(\mathcal{G})$ is labelled by $\Delta_V(L, L')$ iff L and L' are weakly dependent. Edges connecting strongly dependent literals are called *stable*, the remaining ones *unstable*. Global and local variables of $\mathcal{G}$ correspond to those of Π. The label of an unstable edge $e \in \mathcal{E}(\mathcal{G})$ will be denoted by $m(e)$.

As strongly dependent literals remain dependent—whatever a produced substitution looks like—there is no need for checking dependency at run time. No dependency information will therefore be required, and hence the stable edges do not receive a label in a dependency graph. In case of unstable edges, however, we cannot foresee whether literals will stay dependent at run time because dependency solely rests with the actual bindings. This uncertainty is reflected by drawing unstable edges as dashed lines in a graphical representation of dependency graphs. Figure 2 shows a graph $DG_V(\Pi)$ with $V = \{x, y\}$ and $\Pi = \{p(x, z), q(z, y), r(y), s(x)\}$.

Dependency graphs may be viewed as the object code of a *dependency compiler* which analyses syntactical dependencies among body literals of configurations. This compiler will be started every time a new configuration is created but before any substitution is produced. Thereby, a configuration $\mathcal{C} = [\Gamma \mid \Pi]$ is replaced by $\mathcal{C}' = [\Gamma \mid DG_V(\Pi)]$ where V is the global variable set of Π. Now, each time a new substitution is produced the dependency graph can be used by an AND-process to determine shared variable dependencies. In the case of start configurations, which always correspond to program clauses, analysing can be performed before program execution. When compiling a program, every clause $p_0 \leftarrow p_1, \ldots, p_n$, $n \geq 0$, will simply be translated to configuration $[\{p_0\} \mid DG_V(\Pi)]$ with $V = \mathrm{Var}(p_0)$ and $\Pi = \{p_1, \ldots, p_n\}$. As we do not know which head variables will be bound to non-ground terms at run time, we simply take all variables of p_0 as global ones.

Determination of run-time dependencies within a configuration $[\Gamma \mid DG_V(\Pi)]$ is done by a kind of graph reduction, in the course of which dependency tests are performed for literals connected by unstable edges. If their dependency information should turn out to be empty after reduction, the edge is removed to signal independence. Deleting unstable edges finally leads to a reduced dependency graph defined as follows.

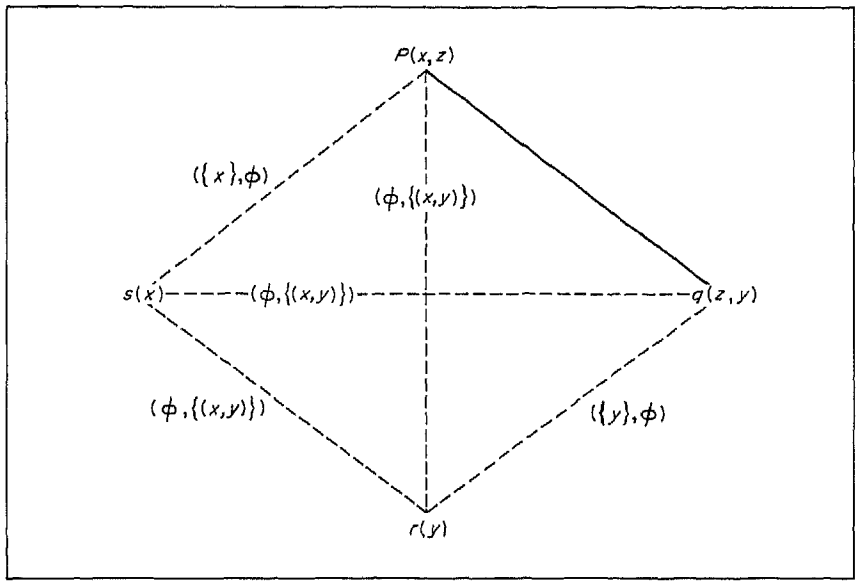

Figure 2. Dependency graph $DG_V(\Pi)$.

DEFINITION 4.3. Let $DG_V(\Pi) = (\mathcal{V}, \mathcal{E})$ be a dependency graph and θ a produced substitution. Then $DG_{V,\theta}(\Pi) = (\mathcal{V}_r, \mathcal{E}_r)$ is the *θ-reduced dependency graph* of Π with

$$\mathcal{V}_r = \mathcal{V},$$

$$\mathcal{E}_r = \{[L, L'] \in \mathcal{E} \mid \Delta_{V,\theta}(L, L') \neq \Delta_\varnothing\}.$$

Figure 3 shows the reduced form of our dependency graph from Figure 2; substitution used for reduction is $\theta = \{x/f(w), y/g(v)\}$.

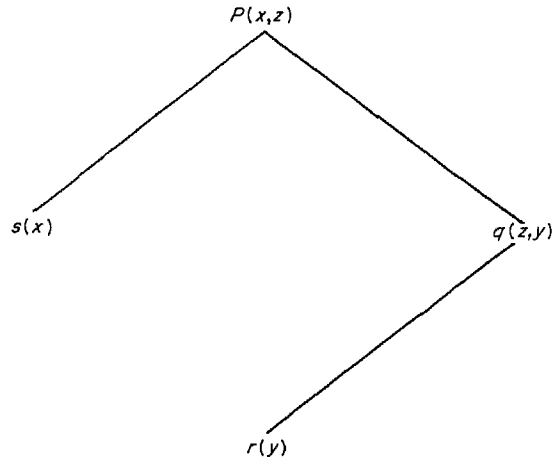

Figure 3. Reduced dependency graph $DG_{V,\theta}(\Pi)$.

Based on reduced dependency graphs determination of independent literals becomes a rather simple task: literals are independent iff they are not connected by edges. The reduced graph of Figure 3, for instance, tells us that $s(x)$ is independent from both $q(z, y)$ and $r(y)$. Further, selecting a set of producer literals—as done in Conery's literal ordering step (Conery, 1987)—may be founded on reduced dependency graphs. Just view literal ordering as a kind of graph searching possibly involving heuristics to find an "optimal" set of independent literals (Schend, 1987). Apart from this, identifying literal ordering with graph searching will allow us to incorporate some well known graph search techniques (Horowitz & Sahni, 1978; Nilsson, 1971) into an implementation of producer/consumer parallelism.

All in all, the concept of reduced dependency graphs has established determination of literal dependencies as a matter of graph reduction. So, in order to investigate efficient algorithms for computing dependencies, we have to find suitable methods for reducing dependency graphs. To achieve this aim a formalism is introduced below to highlight the kind of binding information which is the only relevant one for computing reduced graphs.

5. Binding Patterns

When reducing dependency information and graphs, respectively, we generally do not have to consider term structures of variable values in detail. Obviously, there is no need to know, for instance, which function or variable symbol occurs in which place within a variable binding. The only relevant information required is whether some variable is ground or two variables are connected by a produced substitution. For representing this type of information the concept of binding patterns is introduced below.

DEFINITION 5.1. A *binding pattern* B is a graph $(\mathcal{V}, \mathcal{E})$ with

$$\mathcal{V} \subseteq \mathbf{V} \times \{0, 1\} \text{ such that } (x, m) \in \mathcal{V} \text{ and } (x, n) \in \mathcal{V} \text{ implies } m = n,$$

$$\mathcal{E} \subseteq \{[(x, 0), (y, 0)] \mid (x, 0), (y, 0) \in \mathcal{V}\}.$$

If $\mathcal{V} = \mathcal{E} = \varnothing$, we write $B_\varnothing$ and call it an *empty binding pattern*. The *Carrier* of B is the set $\mathrm{Cr}(B) = [\mathcal{V}(B)]_1$. B is called a *binding pattern for a variable set* V if $\mathrm{Cr}(B) = V$.

When reading the information within a binding pattern the second component of a vertex (x, n) tells us whether variable x is ground ($n = 1$) or not ($n = 0$). Further, an edge $[(x, 0), (y, 0)]$ expresses values of variables x and y to be connected by the produced environment at hand. Some patterns for variable set $V = \{x, y, z\}$ are shown in Figure 4.

According to the following definition every substitution induces a binding pattern on a set of variables.

DEFINITION 5.2. Let V be a finite set of variables and θ a substitution. The binding pattern $BP(V, \theta) = (\mathcal{V}, \mathcal{E})$ with

$$\mathcal{V} = \{(x, n) \mid x \in V,\ n = 1 \text{ iff } \mathrm{Var}(\theta(x)) = \varnothing\},$$

$$\mathcal{E} = \{[(x, 0), (y, 0)] \mid x, y \in V \text{ with } x \text{ connected to } y \text{ by } \theta\},$$

is called the *binding pattern induced by* θ *on* V.

Two induced patterns are given in Figure 5 for variable set $V = \{x, y, z\}$; the corresponding substitutions are $\theta_1 = \{x/u, y/a\}$ and $\theta_2 = \{x/v, y/f(v), z/g(w, a)\}$.

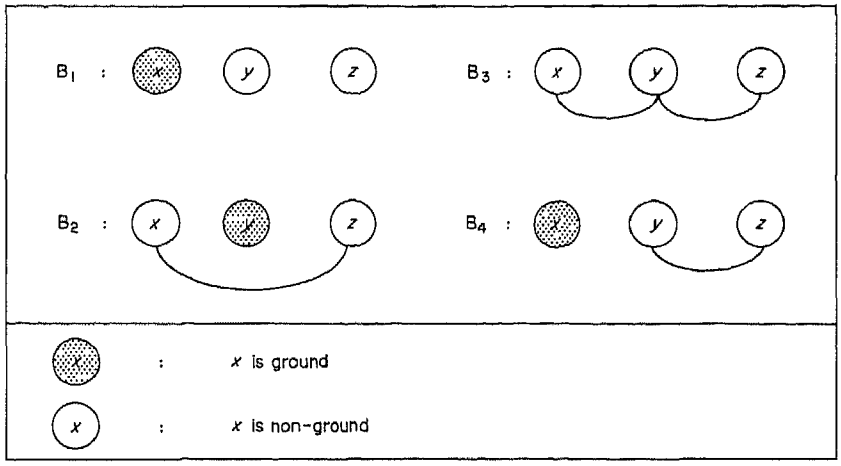

Figure 4. Binding patterns.

Binding patterns do not include any information about local variables of configuration bodies. Such information would be redundant because we always know local variables to be non-ground in a produced environment. With this in mind binding patterns can now be used for reducing dependency information and graphs.

DEFINITION 5.3. If $\Delta_V(L, L') = (D, I)$ is a dependency information and B a binding pattern for V, then $\Delta_{V,B}(L, L')$ is the *B-reduced dependency information* of L and L' consisting of two components D_r and I_r such that

$$D_r = \{x \in D \mid (x, 1) \notin \mathcal{V}(B)\},$$

$$I_r = \{(x, y) \in I \mid [(x, 0), (y, 0)] \in \mathcal{E}(B)\}.$$

In definition of set D_r condition "$(x, 1) \notin \mathcal{V}(B)$" causes local variables to be correctly treated as non-ground. As a consequence every local variable in D will appear in D_r after reduction. Next we extend reduction by binding patterns to dependency graphs.

DEFINITION 5.4. Let $DG_V(\Pi) = (\mathcal{V}, \mathcal{E})$ be a dependency graph and B a binding pattern for V. The *B-reduced dependency graph* of Π is denoted by $DG_{V,B}(\Pi) = (\mathcal{V}_r, \mathcal{E}_r)$ with

$$\mathcal{V}_r = \mathcal{V},$$

$$\mathcal{E}_r = \{[L, L'] \in \mathcal{E} \mid \Delta_{V,B}(L, L') \neq \Delta_\varnothing\}.$$

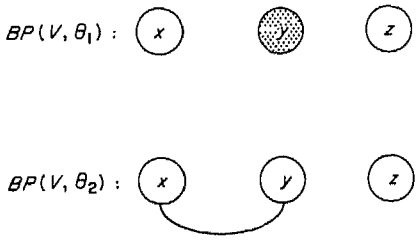

Figure 5. Induced binding patterns.

For any dependency graph $DG_V(\Pi)$ and produced substitution θ one can easily prove $DG_{V,\theta}(\Pi) = DG_{V,BP(V,\theta)}(\Pi)$.

The question now is how can binding patterns be incorporated into dependency checking. An ad hoc answer could be to take a produced substitution, compute the pattern induced on the global variables and then reduce the graph with that pattern. But, as shown below, computing complete binding patterns nearly always leads to information not actually required for reduction. Since this would be in contradiction to our aim of efficient dependency checking, we have to search for a better use of the pattern concept. To convey an idea of how to achieve this aim let us look at an example reduction. Figure 6 shows the dependency graph $\mathcal{G}$ at hand with global variable set $V = \{x, y\}$ and substitution $\theta = \{x/f(b), y/g(u)\}$.

In the course of graph reduction, ground and connection tests will be performed according to the edge labels of graph $\mathcal{G}$. Ground tests are done for variables in D-components and connection tests for variable pairs in the I-components. Suppose we start reduction with edge e_1, the label $(\varnothing, \{(x, y)\})$ of which asks us to perform a connection test for x and y. Because $\mathrm{Var}(\theta(x)) \cap \mathrm{Var}(\theta(y)) = \varnothing$, both variables turn out not to be connected. Thus e_1 is removed from $\mathcal{G}$. In the next reduction step we inspect edge e_2 having the same label as e_1. Again a connection test is executed which, of course, leads to the same result as before. Consequently e_2 is also deleted. Finally, dependency checking is done for e_3 with label $(\{x\}, \varnothing)$. This dependency information requests a ground test for variable x, which itself is bound to the variable free term $f(b)$. Therefore e_3 is also removed from $\mathcal{G}$, so that reduction results in a graph with three nodes $p(x)$, $q(y)$ and $r(x)$ but no edges. All literals finally show mutual independence.

An obvious disadvantage of our reduction method lies in performing ground and connection tests repeatedly for the same variables. This is due to the fact that edge labels of dependency graphs usually show a great degree of conformity which, however, was not taken into account during reduction. But even if we could do graph reduction without

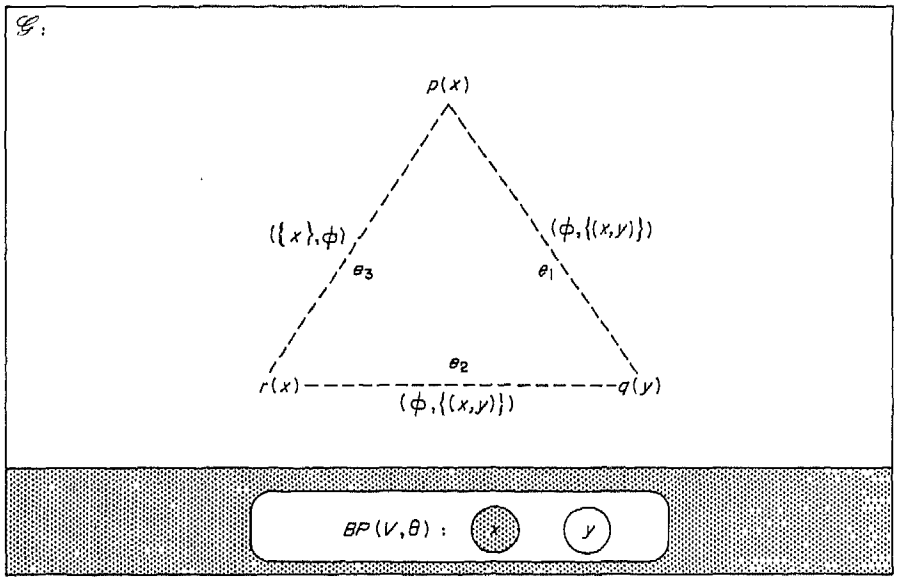

Figure 6. Dependency graph to be reduced by pattern $BP(V, \theta)$.

multiple tests, redundancy may still be met. As an example, for reducing our graph $\mathcal{G}$ of Figure 6 it would have been sufficient to perform a ground test for variable x. From the corresponding result—"x is ground"—we would have been able to infer that x and y are not connected. Every connection test involving variable x was superfluous therefore. All in all, the minimum binding information required for reducing $\mathcal{G}$ is that x is ground. Furthermore, computing the complete pattern $BP(\{x, y\}, \{x/f(b), y/g(u)\})$ before reduction apparently goes along with useless work, as we do not need any information about y.

Concerning superfluous binding tests, we now give a formal characterization of redundancy, and additionally a specification of the minimal binding information required for reducing dependency graphs.

6. Test Collections

A major improvement of our naive reduction method would be to remember test results for multiple use. As a consequence no variable in the D-components nor any variable pair from the I-components had to be tested more than once. For storing results of ground and connection tests, collections of binding patterns will be introduced next. As to give a formal definition of these collections, we first observe that all pieces of binding information gathered during graph reduction are consistent with each other.

DEFINITION 6.1.

(a) Two binding patterns B and B' are *consistent* if the following two conditions hold for all variables x and y.
   (i) $(x, m) \in \mathcal{V}(B)$ and $(x, n) \in \mathcal{V}(B')$ implies $m = n$.
   (ii) If $(x, 0), (y, 0) \in \mathcal{V}(B) \cap \mathcal{V}(B')$, then
   $[(x, 0), (y, 0)] \in \mathcal{E}(B)$ iff $[(x, 0), (y, 0)] \in \mathcal{E}(B')$.

(b) A finite set $\{B_1, \ldots, B_n\}$, $n \geq 0$, of binding patterns is called consistent, if B_i and B_j are consistent for $1 \leq i, j \leq n$.

Consistency of binding patterns guarantees that no contradictory information can be found in them. For instance, if one pattern characterizes a variable to be ground, the other will not express non-groundness (condition (i)). Further, if two variables are connected in one pattern they must also be connected in the other one (condition (ii)), provided it contains information about the variables under consideration. Among the patterns shown in Figure 7, B_1 and B_2 as well as B_1 and B_3 are consistent, whereas B_2 and B_3 are not. B_1 and B_4 are also inconsistent because information about x is contradictory.

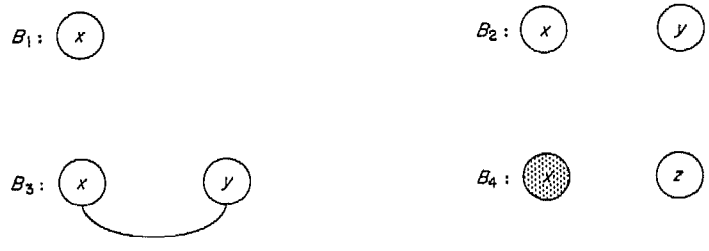

Figure 7. Consistency and inconsistency among binding patterns.

From now on binding patterns are viewed as pieces of binding information, which on their part represent results of binding tests. Grouping such pieces together will lead to what we call a test collection.

DEFINITION 6.2.

(a) A *test collection* $\mathcal{T}$ is a finite, consistent set of binding patterns. An *empty test collection* $\mathcal{T}=\emptyset$ is denoted by $\mathcal{T}_\emptyset$. The set $\mathrm{Cr}(\mathcal{T})=\bigcup_{B\in\mathcal{T}}\mathrm{Cr}(B)$ represents the carrier of $\mathcal{T}$.

(b) If $\mathcal{T}$ is a test collection and $\mathcal{G}$ a dependency graph $DG_V(\Pi)$, then $\mathcal{T}$ is a test collection *for* $\mathcal{G}$ if the carrier of $\mathcal{T}$ contains only global variables, i.e. $\mathrm{Cr}(\mathcal{T})\subseteq V$.

In the following, each time a dependency graph $\mathcal{G}$ and a test collection $\mathcal{T}$ appear together, $\mathcal{T}$ should be considered a test collection *for* $\mathcal{G}$. Before incorporating test collections into reduction of dependency graphs some words should be said about how to interpret information contained in collections like $\mathcal{T}_1$ and $\mathcal{T}_2$ of Figure 8.

Both test collections inform us that variables x, y and z are not ground. In addition, $\mathcal{T}_1$ expresses that y and z are connected, whereas $\mathcal{T}_2$ contains no such information. Finally, neither pattern tells us whether x is connected to y or z. All in all $\mathcal{T}_1$ offers more information than $\mathcal{T}_2$ and therefore may be viewed as an extension of collection $\mathcal{T}_2$.

DEFINITION 6.3. Let B and B' be binding patterns, $\mathcal{T}$ and $\mathcal{T}'$ test collections, respectively.

(a) If B is a subgraph of B', then B' is an *extension* of B denoted by $B\leq B'$. We write $B<B'$ if $B\leq B'$ and $B\neq B'$.

(b) $\mathcal{T}'$ is an *extension* of $\mathcal{T}$, if for each $B\in\mathcal{T}$ there is a $B'\in\mathcal{T}'$ such that B' is an extension of B. As before, we use abbreviations $\mathcal{T}\leq\mathcal{T}'$ and $\mathcal{T}<\mathcal{T}'$.

The definition above induces a partial ordering on test collections. Intuitively, $\mathcal{T}'$ should be regarded *bigger* than $\mathcal{T}$ if $\mathcal{T}'$ contains more information than $\mathcal{T}$. In the following we often use the phrase "extending a test collection" without concerning any special extension. This should simply be understood as adding some piece of binding information resulting from ground and connection tests during graph reduction.

Extension of test collections plays a central role with respect to our aim of developing efficient methods for graph reduction. The basic idea will be to extend an initial test collection, step by step, until it offers sufficient binding information to reduce a graph completely. In this context two questions arise. First, when is a test collection sufficient for reduction, and second, what pieces of information are really necessary for this purpose? To answer these questions we first make clear to what extent arbitrary test collections can be used for reducing dependency graphs.

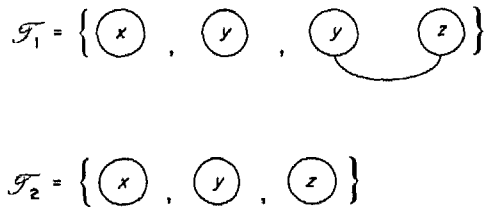

Figure 8. Test collections.

DEFINITION 6.4. Let $\mathcal{T}$ be a test collection.

(a) Two variables x and y are called *disconnected by* $\mathcal{T}$ if one of the following conditions, (i)-(iii), holds for some $B \in \mathcal{T}$.

(i) $(x, 1) \in \mathcal{V}(B)$.

(ii) $(y, 1) \in \mathcal{V}(B)$.

(iii) $(x, 0), (y, 0) \in \mathcal{V}(B)$ and $[(x, 0), (y, 0)] \notin \mathcal{E}(B)$.

(b) Variables x and y are *connected by* $\mathcal{T}$ if $[(x, 0), (y, 0)] \in \mathcal{E}(B)$ for some $B \in \mathcal{T}$.

Obviously two variables are disconnected by a test collection $\mathcal{T}$ if we can definitely infer from $\mathcal{T}$ that their values contain no common variables. But note, being "not disconnected" by a test collection does *not* imply "connected". This discrepancy is due to the fact that $\mathcal{T}$ may contain insufficient information for making a definite statement about connectedness of variables. As an example, looking back to Figure 8 neither $\mathcal{T}_1$ nor $\mathcal{T}_2$ tells us whether x and y are connected or disconnected. Nevertheless test collections can be used for (partially) checking dependency of literals.

DEFINITION 6.5. Let $\Delta_V(L, L') = (D, I)$ be a dependency information. Further, let $\mathcal{T}$ be a test collection. The *$\mathcal{T}$-reduced dependency information* $\Delta_{V,\mathcal{T}}(L, L')$ of L and L' consists of two components D_r and I_r such that

$$D_r = \{x \in D \mid (x, 1) \notin B \text{ for all } B \in \mathcal{T}\},$$

$$I_r = \{(x, y) \in I \mid x \text{ and } y \text{ are } \textit{not} \text{ disconnected by } \mathcal{T}\}.$$

When incorporating test collections into graph reduction, we have to consider that their information content may be insufficient to reduce a graph completely, i.e. to decide for every edge whether it should be deleted or not. However, we can always use a collection to divide the edges into three classes.

DEFINITION 6.6. Let $\mathcal{G}$ be a dependency graph $DG_V(\Pi)$, $[L, L'] \in \mathcal{E}(\mathcal{G})$, and $\mathcal{T}$ a test collection for $\mathcal{G}$.

(a) $[L, L']$ is *stable from* $\mathcal{T}$ if $\Delta_{V,\mathcal{T}'}(L, L') \neq \Delta_\varnothing$ for all extensions $\mathcal{T}'$ of $\mathcal{T}$.

(b) $[L, L']$ is *deleted from* $\mathcal{T}$ if $\Delta_{V,\mathcal{T}}(L, L') = \Delta_\varnothing$.

(c) $[L, L']$ is *unstable at* $\mathcal{T}$ if it is neither stable nor deleted from $\mathcal{T}$.

The set of stable edges is denoted by $SE(\mathcal{G}, \mathcal{T})$, whereas $UE(\mathcal{G}, \mathcal{T})$ denotes the set of unstable edges.

An edge being stable from some collection $\mathcal{T}$ will not be deleted, independent of how much $\mathcal{T}$ is extended to when performing further binding tests. In order to check stability of some edge $[L, L']$, one would principally have to test whether $\Delta_{V,\mathcal{T}'}(L, L') \neq \Delta_\varnothing$ for all extensions $\mathcal{T}'$ of $\mathcal{T}$. Fortunately this can be done in a more simple way without regarding any extension. But first of all, graph reduction by test collections has to be defined.

DEFINITION 6.7. Let $DG_V(\Pi) = (\mathcal{V}, \mathcal{E})$ be a dependency graph and $\mathcal{T}$ a test collection. The *$\mathcal{T}$-reduced dependency graph* $DG_{V,\mathcal{T}}(\Pi) = (\mathcal{V}_r, \mathcal{E}_r)$ of Π is defined by

$$\mathcal{V}_r = \mathcal{V},$$

$$\mathcal{E}_r = \{[L, L'] \in \mathcal{E} \mid \Delta_{V,\mathcal{T}}(L, L') \neq \Delta_\varnothing\}.$$

An edge $[L, L'] \in \mathcal{E}_r$ is labelled by $\Delta_{V,\mathcal{T}}(L, L')$ iff it is unstable at $\mathcal{T}$.

In a graphical representation of a graph $DG_{V,\mathcal{T}}(\Pi)$ all edges being unstable at $\mathcal{T}$ are drawn as dashed lines. This simply reflects that the information in $\mathcal{T}$ was not sufficient for a complete graph reduction. So, further ground and connection tests have to be performed. This also explains why unstable edges are still labelled by reduced dependency information. Figure 9 gives an impression of graph reduction by test collections. The graph shown there has $V=\{x, y, z\}$ as its global variable set.

Obviously, existence of unstable edges in a reduced graph $DG_{V,\mathcal{T}}(\Pi)$ signals incompleteness of $\mathcal{T}$. Based on this observation a test collection may be called complete if there are no unstable edges left after graph reduction.

DEFINITION 6.8. Let $\mathcal{G}$ be a dependency graph. A test collection $\mathcal{T}$ is *reduction complete for* $\mathcal{G}$ if $UE(\mathcal{G}, \mathcal{T})=\emptyset$.

Some examples of complete and incomplete test collections for a graph $\mathcal{G}$ with global variable set $V=\{x, y\}$ are listed in Figure 10. Among test collections shown there $\mathcal{T}_1$, $\mathcal{T}_2$ and $\mathcal{T}_3$ are reduction complete for $\mathcal{G}$; $\mathcal{T}_4$ however is not.

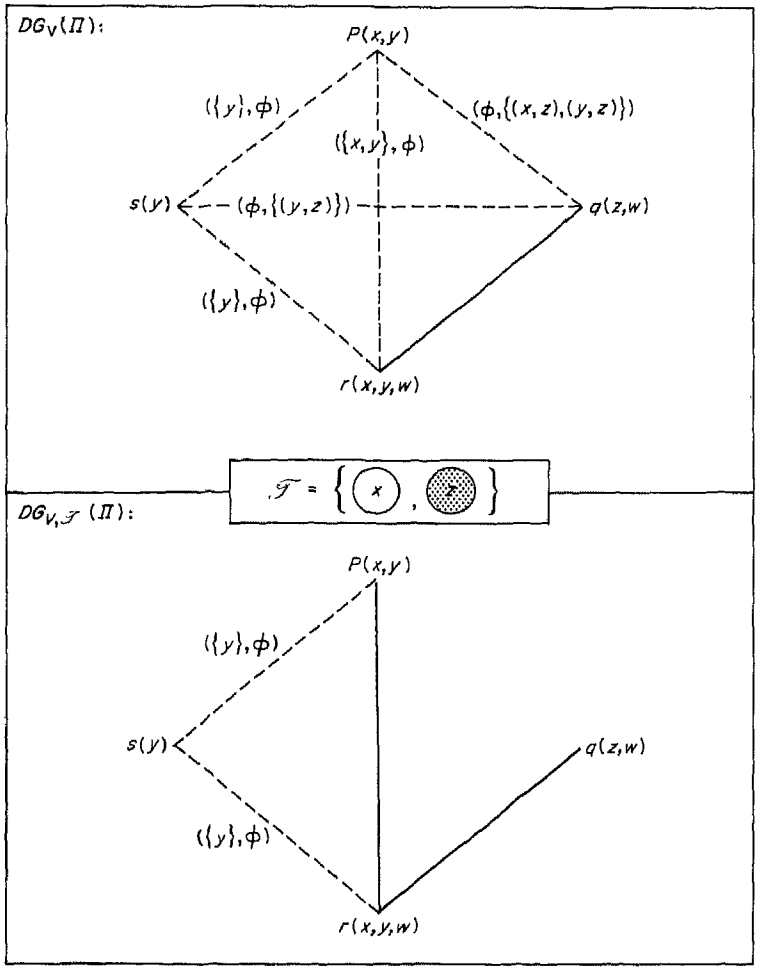

Figure 9. Graph reduction by a test collection $\mathcal{T}$.

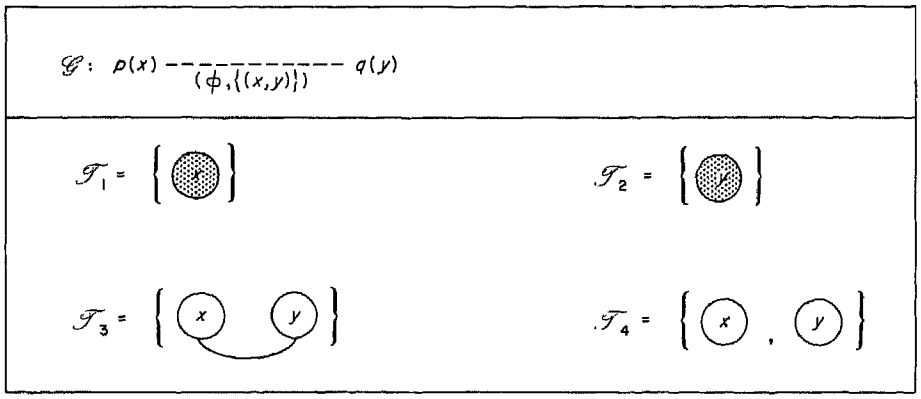

Figure 10. Reduction complete and incomplete test collections for graph $\mathcal{G}$.

After having characterized reduction completeness, we now turn back to the problem of avoiding computation of redundant binding information. In this connection, test collections will be defined which are not only reduction complete, but also show a minimal degree of information. They finally allow the determination of ground and connection tests which are the only necessary ones for reducing dependency graphs completely.

DEFINITION 6.9. Let $\mathcal{G}$ be a dependency graph $DG_V(\Pi)$, θ a produced substitution, and $\mathcal{T}$ a reduction complete test collection for $\mathcal{G}$. Then $\mathcal{T}$ is *minimal reduction complete for* $\mathcal{G}$ if there is no reduction complete test collection $\mathcal{T}'$ for $\mathcal{G}$, such that $\mathcal{T}' < \mathcal{T}$. Further, $\mathcal{T}$ is called *minimal reduction complete for* $\mathcal{G}$ *in accordance with* θ if it is minimal reduction complete and $\mathcal{T} \leq \{BP(V, \theta)\}$.

Figure 11 shows some minimal reduction complete test collections for a dependency graph $\mathcal{G}$ with global variable set $V = \{x, y, z\}$ and substitution $\theta = \{x/f(v), y/u, z/a\}$.

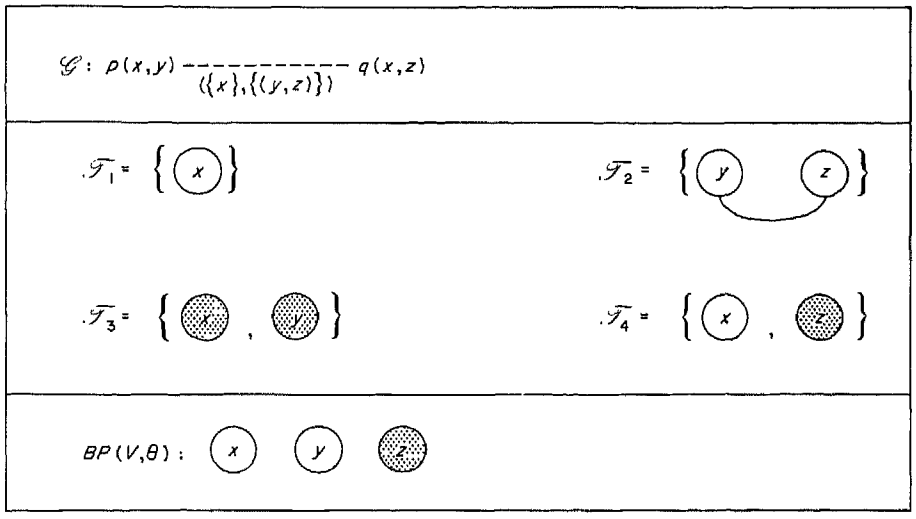

Figure 11. (Non)minimal reduction complete test collections for a graph $\mathcal{G}$.

Among test collections listed there $\mathcal{T}_1$, $\mathcal{T}_2$ and $\mathcal{T}_3$ are minimal reduction complete for $\mathcal{G}$ but $\mathcal{T}_4$ is not. Furthermore, only $\mathcal{T}_1$ is minimal reduction complete in accordance with θ.

Based on test collections which are minimal reduction complete, detection of redundancy goes straightforward. By way of example, collection $\mathcal{T}_1$ of Figure 11 tells us that the information "z is ground" contained in $\mathcal{T}_4$ is actually not required for reducing graph $\mathcal{G}$. Consequently there is no need to perform a ground test for variable z.

Altogether, minimal reduction complete test collections have a declarative character only. They just tell us *what* makes a collection minimal but do not say anything about *how to* compute them when confronted with a produced substitution. So, one question immediately arises: Is it always possible to perform ground and connection tests in a way that computed pieces of binding information constitute a minimal reduction complete collection? The answer, unfortunately, is no as follows from the example reduction of graph $\mathcal{G}$ in Figure 12; substitution used for reduction is $\theta = \{x/a, y/g(u)\}$.

Again, we use the fact that edge labels of a dependency graph represent algorithms for performing binding tests. In this regard label $(\{x, y\}, \varnothing)$ expresses $p(x, y)$ and $q(x, y)$ to be dependent iff variable x or y is non-ground. To check this we eventually have to perform a ground test for x *and* y. However, as mentioned before, the algorithm within a dependency information is non-deterministic; so we have to find by ourselves what to do first: a ground test for x or a ground test for y? According to the order chosen we get one of the test collections $\mathcal{T}_1$ and $\mathcal{T}_2$ of Figure 12. Obviously $\mathcal{T}_2$ is minimal reduction complete in accordance with θ, while $\mathcal{T}_1$ is not even complete. So, if we had decided to test variable x first, we additionally must perform a ground test for y, which finally leads to the test collection $\mathcal{T}_3$ of Figure 12. $\mathcal{T}_3$ is reduction complete but does not have the property of being minimal.

Evidently, choosing the appropriate order of binding tests is essential for avoiding redundancy. The example above, however, already pointed out that non-determinism in dependency information prevents us from performing non-redundant tests only. Therefore, our aim of developing an efficient reduction algorithm cannot be to avoid redundancy completely. We can only aim at keeping test collections computed during graph reduction as small as possible.

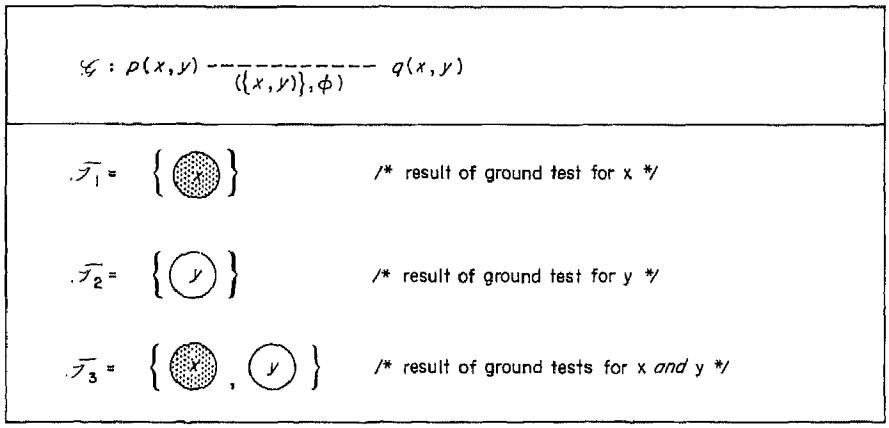

Figure 12. Test collections corresponding to different ordering of binding tests.

7. An Algorithm for Reducing Dependency Graphs

Dependency graphs and test collections have been introduced with a view to developing efficient algorithms for detecting literal dependencies. To convey an idea of how to put the theoretical framework of the preceding sections into practice, an example reduction algorithm will be presented below.

The algorithm itself is based on extension of test collections. Starting with the initial collection $\mathcal{T}_0 = \mathcal{T}_\varnothing$, binding tests are performed guided by dependency information within the graph at hand. Results of these tests are stored in an actual test collection thereby extending it to a more informative one. By this procedure we successively get a sequence $\mathcal{T}_0, \mathcal{T}_1, \ldots, \mathcal{T}_i, \ldots, i \geq 0$, of test collections such that $\mathcal{T}_i \leq \mathcal{T}_{i+1}$. The sequence stops as soon as the actual collection is known to be reduction complete. To check reduction completeness of a test collection the algorithm tries to classify edges as stable or deleted. If all edges can be characterized in this way, no unstable edges are left and the actual collection is reduction complete.

Classifying edges gives rise to the problem of checking stability of an edge efficiently. According to Definition 6.6, an edge $[L, L']$ of a graph $DG_V(\Pi)$ is stable from $\mathcal{T}$ if $\Delta_{V,\mathcal{T}'}(L, L') \neq \Delta_\varnothing$ for all extensions $\mathcal{T}'$ of $\mathcal{T}$. Testing this seems quite expensive but fortunately there are rather simple criterions for determining stability.

THEOREM 7.1. (Stability checking). *Let $\mathcal{G}$ be a dependency graph $DG_V(\Pi)$, $\mathcal{T}$ a test collection for $\mathcal{G}$. An edge $e \in \mathcal{E}(\mathcal{G})$ is stable from $\mathcal{T}$ iff one of the following conditions holds.*

(1) *e connects strongly dependent literals, i.e. has no label.*
(2) *e has label (D, I), and there is a $x \in D$ and a $B \in \mathcal{T}$ such that $(x, 0) \in \mathcal{V}(B)$.*
(3) *e has label (D, I), and there is a $(x, y) \in I$ such that x and y are connected by $\mathcal{T}$.*

PROOF. Remember, all collections and their extensions are test collections *for* $\mathcal{G}$, i.e. their carrier sets do not contain any local variable.

(a) "$\Rightarrow$"
Let $e = [L, L'] \in \mathcal{E}(\mathcal{G})$ be stable from $\mathcal{T}$, i.e. $\Delta_{V,\mathcal{T}'}(L, L') \neq \Delta_\varnothing$ for all extensions $\mathcal{T}'$ of $\mathcal{T}$. Consequently, if $\Delta_V(L, L') = (D, I)$ is the dependency information of L and L', there must be a variable $x \in D$ or a pair $(x, y) \in I$ such that for all extensions $\mathcal{T}'$
(i) $x \in [\Delta_{V,\mathcal{T}'}(L, L')]_1$, or
(ii) $(x, y) \in [\Delta_{V,\mathcal{T}'}(L, L')]_2$.
Now, according to Definition 6.5, (i) requires $(x, 1) \notin \mathcal{V}(B)$ for all $\mathcal{T}'$ and $B \in \mathcal{T}'$. This in return is only guaranteed if x is either a local variable or x is global with $(x, 0) \in \mathcal{V}(B)$ for some $B \in \mathcal{T}$, i.e. is already known to be non-ground by collection $\mathcal{T}$. If x is local, L and L' are strongly dependent, and edge $e = [L, L']$ has no label in $\mathcal{G}$ (condition 1). If x is global and $(x, 0) \in \mathcal{V}(B)$ for some $B \in \mathcal{T}$, we have condition 2. Finally, (ii) can only be true if no extension $\mathcal{T}'$ is able to disconnect x and y. Therefore, $[(x, 0), (y, 0)] \in \mathcal{E}(B)$ for some $B \in \mathcal{T}$ (if not we could easily construct an extension $\mathcal{T}'$ of $\mathcal{T}$, which disconnects x and y). All in all, (ii) implies condition 3.

(b) "$\Leftarrow$"
Let condition 1, 2 or 3 be true for some $e = [L, L'] \in \mathcal{E}(\mathcal{G})$ with $\Delta_V(L, L') = (D, I)$. Further, let $\mathcal{T}'$ be a test collection for $\mathcal{G}$ such that $\mathcal{T} \leq \mathcal{T}'$ and $\Delta_{V,\mathcal{T}'}(L, L') = (D_r, I_r)$. If condition 1 holds, there is a local variable $z \in D$. Since $\mathcal{T}'$ is a collection for $\mathcal{G}$, $z \notin [\mathcal{V}(B)]_1$ for all $B \in \mathcal{T}'$. So we have $z \in D_r$.

If condition 2 holds, there is an $x \in D$ with $(x, 0) \in \mathcal{V}(B)$ for some $B \in \mathcal{T}$. Since $\mathcal{T} \leq \mathcal{T}'$, also $(x, 0) \in \mathcal{V}(B')$ for some $B' \in \mathcal{T}'$, i.e. $x \in D_r$.
If condition 3 holds, there is an $(x, y) \in I$ with $[(x, 0), (y, 0)] \in \mathcal{E}(B)$ for some $B \in \mathcal{T}$. Since $\mathcal{T} \leq \mathcal{T}'$, we also have $[(x, 0), (y, 0)] \in \mathcal{E}(B')$ for some $B' \in \mathcal{T}'$. Consequently, variables x and y are not disconnected by $\mathcal{T}'$, and thus $(x, y) \in I_r$.
As $\mathcal{T}'$ was chosen arbitrarily, we have shown $\Delta_{V,\mathcal{T}'}(L, L') \neq \Delta_\varnothing$ for all extensions $\mathcal{T}'$ of $\mathcal{T}$. So, edge $e = [L, L']$ is stable from $\mathcal{T}$.
Parts (a) and (b) together finally prove our theorem.

With the stability checking method supplied by the theorem above we are now able to formulate our algorithm for reducing dependency graphs. For reasons of clarity it has been kept rather simple and thus should be viewed as a prototype, which needs to be elaborated and adapted to existing or future systems. Due to the relative high costs of connection tests, the algorithm tries to compute a reduction complete test collection solely by performing less expensive ground tests. Connection tests will be delayed until ground tests turn out to result in insufficient information for reducing the graph completely. Therefore our reduction algorithm runs through two distinct phases, in the first of which only D-components of edge labels are considered for guiding reduction. As a consequence only ground tests will be performed. Inspection of I-components and thus performance of connection tests is deferred until the second phase is encountered. The presentation of the algorithm starts with a description of the data, subprocedures and functions used for reduction. An example execution can be found in the Appendix.

Reduction Algorithm
(Algorithm for reducing dependency graphs on the basis of test collections)

Input
$\mathcal{G}$: dependency graph $DG_V(\Pi)$ to be reduced;
θ: a produced substitution;

Output
$\mathcal{G}_r$: reduced dependency graph $DG_{V,\theta}(\Pi)$;

Local Data
Open: set of edges, the labels of which have still to be considered;
$\mathcal{T}$: actual test collection;
$\mathcal{E}_r$: set of edges which were already classified as being stable from $\mathcal{T}$.

Procedures and Functions
Select(*Open*)
This function randomly selects an edge from Open to be classified next.

Gtest($\mathcal{T}$, x, θ)
This procedure checks whether test collection $\mathcal{T}$ tells variable x to be ground or nonground. If no information is available, a ground test is performed for x wrt θ storing its result in $\mathcal{T}$.

Ctest($\mathcal{T}$, (x, y), θ)
First of all this procedure determines whether x and y are connected or disconnected by $\mathcal{T}$. Should be no appropriate information available, a connection test is performed and its result stored in $\mathcal{T}$.

procedure reduce_graph($\mathcal{G}$, θ, $\mathcal{G}_r$);

```
begin
1   𝒯 := ∅;
2   𝒱_r := 𝒱(𝒢);
3   ℰ_r := {e ∈ ℰ(𝒢) | e has no label}; {all edges connecting strongly dependent literals}
4   {first phase: consider D-components of edge labels only}
5   Open := ℰ(𝒢) − ℰ_r;
6   while Open ≠ ∅ do
7       e := Select(Open);
8       D := [m(e)]_1;
9       for all x ∈ D do
10        Gtest(𝒯, x, θ);
11        if (x, 0) ∈ 𝒱(B) for some B ∈ 𝒯 then
12          ℰ_r := ℰ_r ∪ {e};
13          exit; {exit for-loop, continue at line 16}
14        fi;
15      od;
16      Open := Open − {e};
17  od;
18  {second phase: consider I-components of edge labels only}
19  Open := ℰ(𝒢) − ℰ_r;
20  while Open ≠ ∅ do
21      e := Select(Open);
22      I := [m(e)]_2;
23      for all (x, y) ∈ I do
24        Ctest(𝒯, (x, y), θ);
25        if [(x, 0), (y, 0)] ∈ ℰ(B) for some B ∈ 𝒯 then
26          ℰ_r := ℰ_r ∪ {e};
27          exit; {exit for-loop, continue at line 30}
28        fi;
29      od;
30      Open := Open − {e};
31  od;
32  𝒢_r := (𝒱_r, ℰ_r);
end;
```

The input of procedure reduce_graph consists of a dependency graph $\mathcal{G} = DG_V(\Pi)$ and a produced substitution θ. In order to compute the reduced graph $\mathcal{G}_r = DG_{V,\theta}(\Pi)$ it proceeds in two phases, which are implemented as while loops. First of all some initializing is done by assigning the set of stable edges to $\mathcal{E}_r$ (line 3), and the set of unstable edges to Open (line 4). Thereupon phase 1 is entered selecting an edge from Open at a time (line 7), calling procedure Gtest for variables in the actual D-component and checking stability of the edge in line 11. Stable edges are added to $\mathcal{E}_r$ in line 12. Finally the actual edge is removed from Open, thus ensuring termination of the while loop. Phase 2 of our reduction procedure is similar to phase 1 except that I-components of edge labels are inspected. All in all, performing ground and connection tests repeatedly is completely avoided by using procedures Gtest and Ctest. Furthermore, loops of lines 9 and 23 are immediately left by an exit statement as soon as the actual edge is known to be stable. Therefore no more ground and connection tests will be done for elements in D or I not

considered so far. Dividing graph reduction into two phases contributes to minimizing expensive connection tests because a reduction complete test collection might be computed solely by performing ground tests.

As said before, the algorithm above is merely a frame, which can be filled with more elaborated features. For instance, heuristics—like conformity of edge labels—may be used in function *Select* to allow a better ordering of binding tests resulting in less redundancy. Another improvement may be achieved by using *static binding information* for "prereducing" dependency graphs of start configurations at compile time. This information, for example, can be derived from mode declarations or variable annotations (Debray & Warren, 1986; Mellish, 1981; Schend, 1987). Finally, a third possibility for improving efficiency of our reduction algorithm should be mentioned. It is based on the observation that the number of reduction complete test collections for a dependency graph $DG_V(\Pi)$ is finite. Pre-run-time analysis of these collections may help to determine efficient ordering of binding tests prior to dependency checking.

8. Conclusion

For the sake of efficiency most parallel systems realize AND-parallelism in a restricted form only, i.e. literals of a clause body are not solved in parallel unless they are independent. Suitable methods for detecting shared variable dependencies are required, therefore, all the more since dependency checking has to be performed frequently during program execution. Concerning the checking task this paper presented a theoretical framework for analysing and representing such dependencies. One basic idea was to analyse clause literals on a syntactical level for getting a kind of dependency information, which will finally support efficient dependency checking at run time. Analysing literals prior to execution gave rise to syntactical dependencies, which in their turn were represented by dependency graphs. Nodes of such a graph correspond to literals under consideration with edges being labelled by their dependency information. This information can then be used as a guide for efficient dependency checking each time a new substitution is produced. Furthermore, based on dependency graphs determination of independent literals was shown to be a matter of graph reduction. Reduction itself was done by performing ground and connection tests according to the edge labels of the graph.

One major problem arising in connection with graph reduction is redundancy of ground and connection tests. Redundant tests lead to binding information not actually required for producing a reduced graph. In this regard binding patterns and test collections were introduced allowing not only a characterization of non-redundant binding information, but also construction of algorithms for dependency checking. A prototype of such an algorithm was finally presented, which tried to reduce dependency graphs by performing less expensive ground tests only.

The concept of dependency graphs has already been incorporated in a parallel execution system which runs on a transputer hardware (Penner & Klingler, 1988). In addition, application of dependency graphs and test collections is not restricted to detection of shared variable dependencies. For example, literal ordering may be organized on the basis of reduced dependency graphs probably using some well-known graph search procedures and heuristics. Furthermore, generation of execution graph expressions (DeGroot, 1984) can be supported by using the framework presented in this paper. The same applies to a static data dependency analysis, as described by Chang (1985), resulting in what he called data dependency graphs.

References

Chang, J. H., Despain, A. M. (1985). Semi-intelligent backtracking of Prolog based on static data dependency analysis. *Proceedings of the IEEE 1985 International Symposium on Logic Programming*, pp. 10–21.

Chang, J. H. (1983). High performance of Prolog programs based on a static data dependency analysis. Ph.D. Thesis.

Clark, K. L., Gregory, S. (1984). PARLOG: Parallel Programming in Logic. *DOC Report 84/4*, Department of Computing, Imperial College, University of London.

Conery, J. S. (1983). The AND/OR-process model for parallel interpretation of logic programs. Ph.D. Thesis, Technical Report 204, University of California, Irvine.

Conery, J. S. (1987). *Parallel Execution of Logic Programs.* Kluwer Academic Publishers.

Debray, S. K., Warren, D. S. (1986). Automatic mode inference for Prolog programs. *Proceedings of International Symposium on Logic Programming*, IEEE Computer Society Press, pp. 2–11.

DeGroot, D. (1984). Restricted AND-parallelism. *Proceedings of the Intrnational Conference on Fifth Generation Computer Systems*, Tokyo, pp. 471–478.

Halim, Z. (1986). A data-driven machine for OR-parallel evaluation of logic programs. *New Generation Computing* **4**, 5–23.

Hermenegildo, M. V. (1986). An abstract machine for restricted AND-parallel execution of logic programs. *Proceedings of the Third International Conference on Logic Programming*, Springer Verlag, pp. 25–39.

Horowitz, E., Sahni, S. (1978). *Fundamentals of Computer Algorithms.* Computer Science Press.

Ito, N., Onai, R., Masuda, K., Shimuzu, H. (1983). Parallel Prolog machine based on the data flow model. *ICOT Technical Report* **TR-035**, Tokyo.

Kowalski, R. A. (1979). *Logic for Problem Solving. Artificial Intelligence Series*, Vol. 7, New York: Elsevier-North Holland.

Lin, Y.-J., Kumar, V. (1988). An execution model for exploiting AND-parallelism in logic programs. *New Generation Computing* **5**, 393–425.

Lloyd, J. W. (1984). *Foundations of Logic Programming.* Berlin: Springer-Verlag.

Mellish, C. S. (1981). The automatic generation of mode declarations for prolog programs. *DAI Research Paper* **163**, Dept. of Artificial Intelligence, University of Edinburgh.

Nakagawa, H. (1984). AND parallel Prolog with divided assertion set. Proceedings of the IEEE International Symposium on Logic Programming, IEEE Computer Society Press, pp. 22–28.

Nilsson, N. J. (1971). *Problem-Solving Methods in Artificial Intelligence.* New York: McGraw-Hill.

Penner, V., Klingler, A. (1988). *AND Parallel Prolog on Large Scale Transputer Systems.* Parsytec Parallel Systems Technology Ltd. and Institute for Applied Mathematics at University of Aachen, West Germany, Technical Report.

Schend, B. (1987). Ein Modell zur parallelen Ausführung von Logik programmem auf der Basis von partiellen AND-Prozessen. Ph.D. Thesis.

Schend, B. (1989). A sequential view of AND-parallelism through partial AND-processes. *Proceedings of 1989 International Joint Conference on Artificial Intelligence*, pp. 163–169.

Seungbeom, K., Seungryoul, M., Jung, W. C. (1986). A parallel execution model of logic program based on dependency relationship graph. *Proceedings of the 1986 IEEE International Conference on Parallel Processing*, St. Charles, Illinois, pp. 976–991.

Smith, D. E., Genesereth, M. R. (1985). Odering conjunctive queries. *Artificial Intelligence* **26**, 171–215.

Tick, E., Warren, D. H. D. (1984). Towards a pipelined prolog processor. *Proceedings of the IEEE International Symposium on Logic Programming*, IEEE Computer Society Press, pp. 29–40, Atlantic City.

Tung, Y.-W., Moldovan, D. I. (1986). Detection of AND-parallelism in logic programming. *Proceedings of the 1986 IEEE International Conference on Parallel Processing*, St. Charles, Illinois, pp. 19–22.

Umeyama, S., Tamura, K. (1983). A parallel execution model of logic programs. *The 10th Annual International Symposium on Computer Architecture*, ACM, pp. 349–355.

Appendix

Example execution of the reduction algorithm.

Reducing the graph $\mathcal{G}$ from Figure 6 with substitution $\theta = \{x/f(b), y/g(u)\}$ starts with

$\mathcal{T} = \emptyset$,

$\mathcal{V}_r = \mathcal{V}(\mathcal{G}) = \{p(x), r(x), q(y)\}$,

$\mathcal{E}_r = \{e \in \mathcal{E}(\mathcal{G}) \mid e \text{ has no label}\} = \emptyset$,

Open $= \mathcal{E}(\mathcal{G}) - \mathcal{E}_r = \{e_1, e_2, e_3\}$;

Upon entering the first while loop assume e_3 with D-component $[m(e_3)]_1 = \{x\}$ to be selected in line 7. After selection procedure Gtest is called in line 10 for checking

groundness of x. As no information about x is available in $\mathcal{T}$ at that time, a ground test has to be performed. Its result is stored in $\mathcal{T}$, which now gets

$$\mathcal{T} = \{\circledast\}.$$

Next, since the if-condition in line 11 evaluates to false, we directly jump to line 16 removing e_3 from Open. The remaining edges e_1 and e_2 show empty D-components. As a consequence, the for loop of line 9 is skipped, and removing both edges finally makes Open empty. The first reduction phase is finished now; no stable edges have been found so far.

Before starting the second phase, Open is set to $\mathscr{E}(\mathcal{G}) - \mathscr{E}_r$, which again is $\{e_1, e_2, e_3\}$. Whenever an edge with an empty I-component, like e_3, is selected in line 21 it is immediately removed from Open in line 30. So, let us first look at e_1 where $[m(e_1)]_2 = \{(x, y)\}$. Procedure Ctest is invoked for checking connectedness of x and y. First, it reads the actual test collection $\mathcal{T}$, which tells variable x to be ground. Consequently x and y cannot be connected. The for loop in line 23 is not executed, and e_1 is removed from Open in line 30. This also happens to e_2 since having the same label as e_1. After finishing the second phase we still have $\mathscr{E}_r = \varnothing$. So, the reduced graph $\mathcal{G}_r$ is set to $(\mathscr{V}_r, \varnothing)$ in line 32, reflecting that all nodes (literals) are mutually independent.